Table of Contents

Building Database Objects with Common Language Runtime (CLR) Integration

3/24/2017 • 2 min to read • Edit Online

You can build database objects using the SQL Server integration with the .NET Framework common language runtime (CLR). Managed code that runs inside of Microsoft SQL Server is referred to as a "CLR routine." These routines include:

- Scalar-valued user-defined functions (scalar UDFs)
- Table-valued user-defined functions (TVFs)
- User-defined procedures (UDPs)
- User-defined triggers

  CLR routines have the same structure in managed code. They are mapped to public, static (shared in Microsoft Visual Basic .NET) methods of a class. In addition to routines, user-defined types (UDTs) and user-defined aggregate functions can also be defined using the .NET Framework. UDTs and user-defined aggregates are mapped to entire .NET Framework classes.

  Each type of .NET Framework routine has a Transact-SQL declaration and can be used anywhere in SQL Server that the Transact-SQL equivalent can be used. For instance, scalar UDFs can be used in any scalar expression. A TVF can be used in any FROM clause. A procedure can be invoked in an EXEC statement or invoked from a client application.

> **NOTE**
>
> Execution of a CLR object (user-defined function, user-defined type, or trigger) on the common language runtime can take place on multiple threads (parallel plan), if the query optimizer decides it is beneficial. However, if a user-defined function accesses data, execution will be on a serial plan. When executed on a server version prior to SQL Server 2008, if a user-defined function contains LOB parameters or return values, execution also must be on a serial plan.

The following table lists the topics covered in this section.

Getting Started with CLR Integration
Provides a brief overview of the libraries and namespaces required to compile object using CLR integration with SQL Server. Includes an example "Hello World" CLR stored procedure.

Supported .NET Framework Libraries
Provides information on the .NET Framework libraries supported by CLR integration.

CLR Integration Programming Model Restrictions
Provides information about CLR integration programming model restrictions.

SQL Server Data Types in the .NET Framework
An overview of SQL Server data types and their .NET Framework equivalents.

Overview of CLR Integration Custom Attributes
Provides information about CLR integration custom attributes.

CLR User-Defined Functions
Describes how to implement and use the various types of CLR functions: table-valued, scalar, and user-defined

aggregate functions.

CLR User-Defined Types
Describes how to implement and use CLR user-defined types.

CLR Stored Procedures
Describes how to implement and use CLR stored procedures.

CLR Triggers
Describes how to implement and use CLR triggers.

See Also

Common Language Runtime (CLR) Integration Overview

SQL Server Data Types in the .NET Framework

3/24/2017 • 1 min to read • Edit Online

The **SqlTypes** library is part of the base class library of the Microsoft .NET Framework. It is designed to provide data types with the same semantics and precision as those found in the SQL Server database. This topic describes the new semantics to .NET Framework programmers, and introduces the types implemented in the **System.Data.SqlTypes** namespace that is included in the **System.Data** library.

This following table lists the topics in this section.

Nullability and Three-Value Logic Comparisons
Discusses how NULL values are handled with common language runtime (CLR) integration data types.

Collation and CLR Integration Data Types
Describes how collations are handled with CLR integration.

Handling Large Object (LOB) Parameters in the CLR
Describes how to pass LOB types between SQL Server and the CLR.

Mapping CLR Parameter Data
Shows data type mappings between SQL Server, CLR integration, and the .NET Framework.

CLR User-Defined Functions

3/24/2017 • 1 min to read • Edit Online

User-defined functions are routines that can take parameters, perform calculations or other actions, and return a result. Beginning with SQL Server 2005, you can write user-defined functions in any Microsoft .NET Framework programming language, such as Microsoft Visual Basic .NET or Microsoft Visual C#.

There are two types of functions: scalar, which returns a single value, and table-valued, which returns a set of rows.

The following table lists the topics in this section.

CLR Scalar-Valued Functions
Covers implementation requirements and examples of scalar-valued functions.

CLR Table-Valued Functions
Discusses how to implement and use table-valued functions (TVFs), as well as differences between Transact-SQL and common language runtime (CLR) TVFs.

CLR User-Defined Aggregates
Describes how to implement and use user-defined aggregates.

See Also

User-Defined Functions

CLR User-Defined Types

3/24/2017 • 2 min to read • Edit Online

SQL Server gives you the ability to create database objects that are programmed against an assembly created in the.NET Framework common language runtime (CLR). Database objects that can take advantage of the rich programming model provided by the CLR include triggers, stored procedures, functions, aggregate functions, and types.

> **NOTE**
> The ability to execute CLR code is set to OFF by default in SQL Server. The CLR can be enabled by using the **sp_configure** system stored procedure.

Beginning with SQL Server 2005, you can use user-defined types (UDTs) to extend the scalar type system of the server, enabling storage of CLR objects in a SQL Server database. UDTs can contain multiple elements and can have behaviors, differentiating them from the traditional alias data types which consist of a single SQL Server system data type.

Because UDTs are accessed by the system as a whole, their use for complex data types may negatively impact performance. Complex data is generally best modeled using traditional rows and tables. UDTs in SQL Server are well suited to the following:

- Date, time, currency, and extended numeric types
- Geospatial applications
- Encoded or encrypted data

The process of developing UDTs in SQL Server consists of the following steps:

1. **Code and build the assembly that defines the UDT.** UDTs are defined using any of the languages supported by the.NET Framework common language runtime (CLR) that produce verifiable code. This includes Visual C# and Visual Basic .NET. The data is exposed as fields and properties of a .NET Framework class or structure, and behaviors are defined by methods of the class or structure.
2. **Register the assembly.** UDTs can be deployed through the Visual Studio user interface in a database project, or by using the Transact-SQL CREATE ASSEMBLY statement, which copies the assembly containing the class or structure into a database.
3. **Create the UDT in SQL Server.** Once an assembly is loaded into a host database, you use the Transact-SQL CREATE TYPE statement to create a UDT and expose the members of the class or structure as members of the UDT. UDTs exist only in the context of a single database, and, once registered, have no dependencies on the external files from which they were created.

   > **NOTE**
   > Before SQL Server 2005, UDTs created from .NET Framework assemblies were not supported. However, you can still use SQL Server alias data types by using **sp_addtype**. The CREATE TYPE syntax can be used for creating both native SQL Server user-defined data types and UDTs.

4. **Create tables, variables, or parameters using the UDT** Beginning with SQL Server 2005, a user-defined type can be used as the column definition of a table, as a variable in a Transact-SQL batch, or as an

argument of a Transact-SQL function or stored procedure.

In This Section

Creating a User-Defined Type
Describes how to create UDTs.

Registering User-Defined Types in SQL Server
Describes how to register and manage UDTs in SQL Server.

Working with User-Defined Types in SQL Server
Describes how to create queries using UDTs.

Accessing User-Defined Types in ADO.NET
Describes how to work with UDTs using the .NET Framework Data Provider for SQL Server in ADO.NET.

CLR Integration Custom Attributes for CLR Routines

3/24/2017 • 1 min to read • Edit Online

The attributes listed can be applied to common language runtime (CLR) routines, user-defined types, and user-defined aggregates that are registered in Microsoft SQL Server. If the attribute is not applied, SQL Server assumes the default value. The attributes listed are defined in the **Microsoft.SqlServer.Server** namespace.

The SqlUserDefinedAggregate Attribute

The **SqlUserDefinedAggregate** attribute indicates that the method should be registered as a user-defined aggregate. Every user-defined aggregate must be annotated with this attribute.

For more information, see SqlUserDefinedAggregateAttribute.

The SqlFunction Attribute

The **SqlFunction** attribute indicates the method should be registered as a function, with the appropriate function attributes set.

For more information, see SqlFunctionAttribute.

The SqlFacet Attribute

The **SqlFacet** attribute is used to return information about the return type of a user-defined type (UDT) expression.

For more information, see SqlFacetAttribute.

The SqlProcedure Attribute

The **SqlProcedure** attribute indicates the method should be registered as a stored procedure. This attribute is used only by Visual Studio to register the specified method as a stored procedure automatically; it is not used by SQL Server.

For more information, see SqlProcedureAttribute.

The SqlTrigger Attribute

The **SqlTrigger** attribute indicates the method should be registered as a trigger.

For more information, see SqlTriggerContext and SqlTriggerAttribute.

The SqlUserDefinedTypeAttribute

You can apply the SqlUserDefinedTypeAttribute to a class definition in the assembly. It causes SQL Server to create a user-defined type that is bound to the class definition which has this custom attribute.

For more information, see SqlUserDefinedTypeAttribute.

The SqlMethod Attribute

The **SqlMethod** attribute is used to indicate the determinism and data access properties of a method or a property on a UDT.

For more information, see SqlMethodAttribute.

See Also

CLR User-Defined Aggregates
CLR User-Defined Functions
CLR User-Defined Types
CLR Stored Procedures
CLR Triggers

CLR Integration Programming Model Restrictions

3/24/2017 • 2 min to read • Edit Online

When you are building a managed stored procedure or other managed database object, there are certain code checks performed by SQL Server that need to be considered. SQL Server performs checks on the managed code assembly when it is first registered in the database, using the **CREATE ASSEMBLY** statement, and also at runtime. The managed code is also checked at runtime because in an assembly there may be code paths that may never actually be reached at runtime. This provides flexibility for registering third party assemblies, especially, so that an assembly wouldn't be blocked where there is 'unsafe' code designed to run in a client environment but would never be executed in the hosted CLR. The requirements that the managed code must meet depend on whether the assembly is registered as **SAFE**, **EXTERNAL_ACCESS**, or **UNSAFE**, **SAFE** being the strictest, and are listed below.

In addition to restrictions being placed on the managed code assemblies, there are also code security permissions that are granted. The common language runtime (CLR) supports a security model called code access security (CAS) for managed code. In this model, permissions are granted to assemblies based on the identity of the code. **SAFE**, **EXTERNAL_ACCESS**, and **UNSAFE** assemblies have different CAS permissions. For more information, see CLR Integration Code Access Security.

CREATE ASSEMBLY Checks

When the **CREATE ASSEMBLY** statement is run, the following checks are performed for each security level. If any check fails, **CREATE ASSEMBLY** will fail with an error message.

Global (any security level)

All referenced assemblies must meet one or more of the following criteria:

- The assembly is already registered in the database.
- The assembly is one of the supported assemblies. For more information, see Supported .NET Framework Libraries.
- You are using **CREATE ASSEMBLY FROM**<*location*>, and all the referenced assemblies and their dependencies are available in <*location*>.
- You are using **CREATE ASSEMBLY FROM**<*bytes* ... >, and all the references are specified via space separated bytes.

EXTERNAL_ACCESS

All **EXTERNAL_ACCESS** assemblies must meet the following criteria:

- Static fields are not used to store information. Read-only static fields are allowed.
- PEVerify test is passed. The PEVerify tool (peverify.exe), which checks that the MSIL code and associated metadata meet type safety requirements, is provided with the .NET Framework SDK.
- Synchronization, for example with the **SynchronizationAttribute** class, is not used.
- Finalizer methods are not used.

  The following custom attributes are disallowed in **EXTERNAL_ACCESS** assemblies:

- System.ContextStaticAttribute
- System.MTAThreadAttribute

- System.Runtime.CompilerServices.MethodImplAttribute
- System.Runtime.CompilerServices.CompilationRelaxationsAttribute
- System.Runtime.Remoting.Contexts.ContextAttribute
- System.Runtime.Remoting.Contexts.SynchronizationAttribute
- System.Runtime.InteropServices.DllImportAttribute
- System.Security.Permissions.CodeAccessSecurityAttribute
- System.Security.SuppressUnmanagedCodeSecurityAttribute
- System.Security.UnverifiableCodeAttribute
- System.STAThreadAttribute
- System.ThreadStaticAttribute

SAFE

- All **EXTERNAL_ACCESS** assembly conditions are checked.

Runtime Checks

At runtime, the code assembly is checked for the following conditions. If any of these conditions are found, the managed code will not be allowed to run and an exception will be thrown.

UNSAFE

Loading an assembly—either explicitly by calling the **System.Reflection.Assembly.Load()** method from a byte array, or implicitly through the use of **Reflection.Emit** namespace—is not permitted.

EXTERNAL_ACCESS

All **UNSAFE** conditions are checked.

All types and methods annotated with the following host protection attribute (HPA) values in the supported list of assemblies are disallowed.

- SelfAffectingProcessMgmt
- SelfAffectingThreading
- Synchronization
- SharedState
- ExternalProcessMgmt
- ExternalThreading
- SecurityInfrastructure
- MayLeakOnAbort
- UI

  For more information about HPAs and a list of disallowed types and members in the supported assemblies, see Host Protection Attributes and CLR Integration Programming.

SAFE

All **EXTERNAL_ACCESS** conditions are checked.

See Also

Supported .NET Framework Libraries
CLR Integration Code Access Security
Host Protection Attributes and CLR Integration Programming
Creating an Assembly

Getting Started with CLR Integration

3/24/2017 • 4 min to read • Edit Online

This topic provides an overview of the namespaces and libraries required to compile database objects using the Microsoft SQL Server integration with the .NET Framework common language runtime (CLR). The topic also shows you how to write, compile, and run a simple CLR stored procedure written in Microsoft Visual C#.

Required Namespaces

The components required to develop basic CLR database objects are installed with SQL Server. CLR integration functionality is exposed in an assembly called system.data.dll, which is part of the .NET Framework. This assembly can be found in the Global Assembly Cache (GAC) as well as in the .NET Framework directory. A reference to this assembly is typically added automatically by both command line tools and Microsoft Visual Studio, so there is no need to add it manually.

The system.data.dll assembly contains the following namespaces, which are required for compiling CLR database objects:

`System.Data`

`System.Data.Sql`

`Microsoft.SqlServer.Server`

`System.Data.SqlTypes`

Writing A Simple "Hello World" Stored Procedure

Copy and paste the following Visual C# or Microsoft Visual Basic code into a text editor, and save it in a file named "helloworld.cs" or "helloworld.vb".

```
using System;
using System.Data;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;

public class HelloWorldProc
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void HelloWorld(out string text)
    {
        SqlContext.Pipe.Send("Hello world!" + Environment.NewLine);
        text = "Hello world!";
    }
}
```

```
Imports System
Imports System.Data
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlTypes
Imports System.Runtime.InteropServices

Public Class HelloWorldProc
    \<Microsoft.SqlServer.Server.SqlProcedure> _
    Public Shared  Sub HelloWorld(\<Out()> ByRef text as String)
        SqlContext.Pipe.Send("Hello world!" & Environment.NewLine)
        text = "Hello world!"
    End Sub
End Class
```

This simple program contains a single static method on a public class. This method uses two new classes, **SqlContext** and **SqlPipe**, for creating managed database objects to output a simple text message. The method also assigns the string "Hello world!" as the value of an out parameter. This method can be declared as a stored procedure in SQL Server, and then run in the same manner as a Transact-SQL stored procedure.

Compile this program as a library, load it into SQL Server, and run it as a stored procedure.

Compile the "Hello World" stored procedure

SQL Server installs the Microsoft .NET Framework redistribution files by default. These files include csc.exe and vbc.exe, the command-line compilers for Visual C# and Visual Basic programs. In order to compile our sample, you must modify your path variable to point to the directory containing csc.exe or vbc.exe. The following is the default installation path of the .NET Framework.

```
C:\Windows\Microsoft.NET\Framework\(version)
```

Version contains the version number of the installed .NET Framework redistributable. For example:

```
C:\Windows\Microsoft.NET\Framework\v4.6.1
```

Once you have added the .NET Framework directory to your path, you can compile the sample stored procedure into an assembly with the following command. The **/target** option allows you to compile it into an assembly.

For Visual C# source files:

```
csc /target:library helloworld.cs
```

For Visual Basic source files:

```
vbc /target:library helloworld.vb
```

These commands launch the Visual C# or Visual Basic compiler using the /target option to specify building a library DLL.

Loading and Running the "Hello World" Stored Procedure in SQL Server

Once the sample procedure has successfully compiled, you can test it in SQL Server. To do this, open SQL Server Management Studio and create a new query, connecting to a suitable test database (for example, the

AdventureWorks sample database).

The ability to execute common language runtime (CLR) code is set to OFF by default in SQL Server. The CLR code can be enabled by using the **sp_configure** system stored procedure. For more information, see Enabling CLR Integration.

We will need to create the assembly so we can access the stored procedure. For this example, we will assume that you have created the helloworld.dll assembly in the C:\ directory. Add the following Transact-SQL statement to your query.

```
CREATE ASSEMBLY helloworld from 'c:\helloworld.dll' WITH PERMISSION_SET = SAFE
```

Once the assembly has been created, we can now access our HelloWorld method by using the create procedure statement. We will call our stored procedure "hello":

```
CREATE PROCEDURE hello
@i nchar(25) OUTPUT
AS
EXTERNAL NAME helloworld.HelloWorldProc.HelloWorld
-- if the HelloWorldProc class is inside a namespace (called MyNS),
-- the last line in the create procedure statement would be
-- EXTERNAL NAME helloworld.[MyNS.HelloWorldProc].HelloWorld
```

Once the procedure has been created, it can be run just like a normal stored procedure written in Transact-SQL. Execute the following command:

```
DECLARE @J nchar(25)
EXEC hello @J out
PRINT @J
```

This should result in the following output in the SQL Server Management Studio messages window.

```
Hello world!
Hello world!
```

Removing the "Hello World" Stored Procedure Sample

When you are finished running the sample stored procedure, you can remove the procedure and the assembly from your test database.

First, remove the procedure using the drop procedure command.

```
IF EXISTS (SELECT name FROM sysobjects WHERE name = 'hello')
   drop procedure hello
```

Once the procedure has been dropped, you can remove the assembly containing your sample code.

```
IF EXISTS (SELECT name FROM sys.assemblies WHERE name = 'helloworld')
   drop assembly helloworld
```

See also

Supported .NET Framework Libraries

3/24/2017 • 2 min to read • Edit Online

With the common language runtime (CLR) hosted in SQL Server, you can author stored procedures, triggers, user-defined functions, user-defined types, and user-defined aggregates in managed code. With the functionality found in the .NET Framework class libraries, you have access to pre-built classes that provide functionality for string manipulation, advanced math operations, file access, cryptography, and more. These classes can be accessed from any managed stored procedure, user-defined type, trigger, user-defined function, or user-defined aggregate.

> **NOTE**
>
> If you service or upgrade unsupported assemblies in the global assembly cache (GAC), your SQL Server application might stop working. This is because servicing or upgrading libraries in the GAC does not update those assemblies inside SQL Server. If an assembly exists both in a SQL Server database and in the GAC, the two copies of the assembly must exactly match. If they do not match, an error will occur when the assembly is used by SQL Server CLR integration. If you service or upgrade any assemblies in the GAC that are also registered in the database, including unsupported .NET Framework assemblies, make sure to also service or upgrade the copy of the assembly inside your SQL Server databases with the **ALTER ASSEMBLY** statement. For more information, see Knowledge Base article 949080.

Supported Libraries

Beginning with SQL Server 2005, SQL Server has a list of supported .NET Framework libraries, which have been tested to ensure that they meet reliability and security standards for interaction with SQL Server. Supported libraries do not need to be explicitly registered on the server before they can be used in your code; SQL Server loads them directly from the Global Assembly Cache (GAC).

The libraries/namespaces supported by CLR integration in SQL Server are:

- CustomMarshalers
- Microsoft.VisualBasic
- Microsoft.VisualC
- mscorlib
- System
- System.Configuration
- System.Data
- System.Data.OracleClient
- System.Data.SqlXml
- System.Deployment
- System.Security
- System.Transactions
- System.Web.Services
- System.Xml

- System.Core.dll
- System.Xml.Linq.dll

Unsupported Libraries

Unsupported libraries can still be called from your managed stored procedures, triggers, user-defined functions, user-defined types, and user-defined aggregates. The unsupported library must first be registered in the SQL Server database, using the **CREATE ASSEMBLY** statement, before it can be used in your code. Any unsupported library that is registered and run on the server should be reviewed and tested for security and reliability.

For example, the **System.DirectoryServices** namespace is not supported. You must register the System.DirectoryServices.dll assembly with **UNSAFE** permissions before you can call it from your code. The **UNSAFE** permission is necessary because classes in the **System.DirectoryServices** namespace do not meet the requirements for **SAFE** or **EXTERNAL_ACCESS**. For more information, see CLR Integration Programming Model Restrictions and CLR Integration Code Access Security.

See Also

Creating an Assembly
CLR Integration Code Access Security
CLR Integration Programming Model Restrictions